Student Name: **Student id:** **Sect: #:**

QUESTION #	1	2	3	4	5	6	TOTAL
MAX POINTS	8	12	10	10	11	12	63
POINTS EARNED							

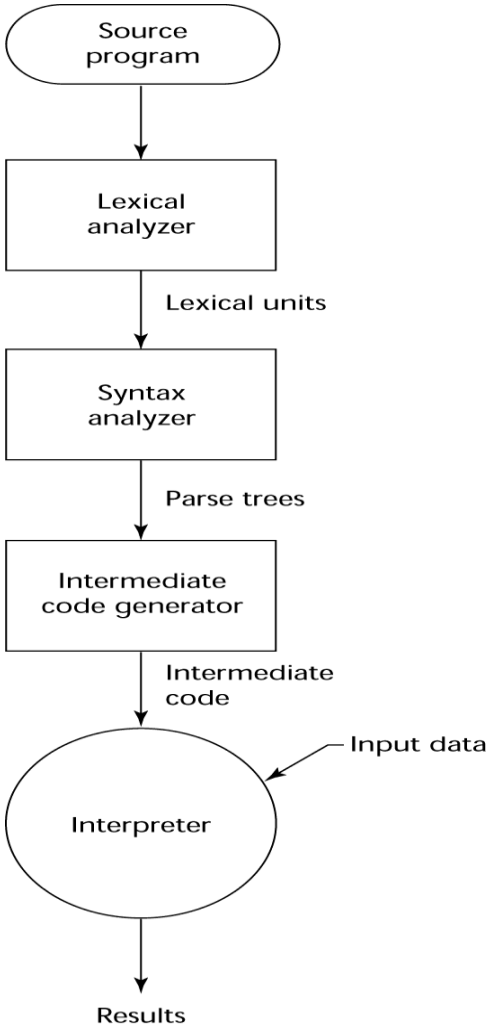
University of Bahrain College of Information Technology
Department of Computer Science

ITCS 332: Organization of Programming Languages FIRST TEST Date: APR 06, 2016

****

QUESTION *ONE*: [8 pts]

Draw the flowchart that illustrates the steps of a hybrid implementation system of any HLL .



QUESTION *TWO*: Fill in blanks [12 pts]

1) Give two features of imperative languages.

- **Imperative languages are based on assignment and variables.**
- **Algorithms in imperative languages consist of detailed ordered steps.**

2) Give two suggestions to the programmer to reduce the compilation times in C++ language.

- **Use operands of the same type in expressions, if possible.**
- **Use explicit casting for types.**
- **Use subprograms instead of macros**

3) Give two suggestions to the programmer to reduce the execution times in C++ language.

- **Use operands of the same type in expressions, if possible.**
- **Use explicit casting for types.**
- **Use macros instead of subprograms.**

4) Give two suggestions to the programmer to improve the readability of a language.

- **Use meaningful names.**
- **Use familiar language constructs or avoid using unfamiliar ones.**
- **Reduce the use of overload operators.**
- **Reduce the use of aliases.**

5) Give two suggestions to the language designer to improve the reliability of a language .

- **Support "natural" ways of expressing algorithms.**
- **Exclude aliasing from the language.**
- **Include extensive capabilities for exception handling.**
- **Remove pointers from the language.**

6) Give two advantages of using preprocessor macros:

- **Programs with macros are generic and flexible (Macros use Typeless parameters).**
- **Programs with macros are fast in execution.**

QUESTION THREE: **[10 pts]**

- Convert each of the following BNF rules into ONE equivalent EBNF rule

1) *<run> → <leg> | <run> "#" <leg> | <run> "!" <leg> | <run> "$" <leg>*

<run> → <leg> { (# | ! | $) <leg> }

2) *<int> → <decs> # | <sign> <decs> #*
<decs> → <digit> | <decs> <digit>
<sign> → + | –

<int> → [(+|-)] <digit> { <digit> } #

3) *<real> → <sign> <decs> "." <decs> "E" <decs> | <sign> <decs> "." <decs>*
| <sign> <decs> "." <decs> "E" <sign> <decs>
<decs> → <dig> | <decs> <dig>
<sign> → "+" | "–"

<real> → (+|-) <dig>{<dig>} "." <dig>{<dig>} ["E" [(+|-)]
<dig>{<dig>}]

- **Fill in blanks**

1) Rewrite the rule: <run> → <leg> … given above, so that operators # ! $ associate right to left.

<run> → <leg> | <leg> "#" <run> | <leg> "!" <run> | <leg> "$" <run>

2) In axiomatic semantics, each statement is preceded by a **precondition** and followed by a **postcondition**.

3) Name two issues (examples) in programming languages that cannot be described by BNF rules:

- **Type compatibility rules.**
- **Identifiers must be declared before referenced.**

QUESTION FOUR: **[10 pts]**

- **Fill in blanks**

1) The weakest precondition for each kind of statements can be computed by **an axiom** or by **an inference rule**.
2) In denotational semantics, the state changes are defined by **MATHEMATICAL FUNCTIONS**. In operational semantics, the state changes are defined by **CODED ALGORITHMS.**
3) The number of lexemes in a C++ statement: **`if(!(xt[j]-9)) xy=ct;`** is **16** and the number tokens of is **11**.
4) Ambiguities in BNF grammars of arithmetic expression can be solved by introducing **precedence rules** and **assosiativity rule** into its grammar.

- Convert each of the following EBNF rule into equivalent BNF rule(s).

1) `<par>` → `<sent>` { ; `<sent>` }

<par> → <sent>
| <sent> ; < par >

2) `<call>` → `<id>` "(" `<parm>` { "#" `<parm>` } ")"

<call> → <id> "(" <parms> ")"
<parms> → <parm>
| <parm> "#" <parms>

3) `<isa>` → `<int>` { (! | :) `<int>` }

<isa> → <int>
| <int> ! <isa>
| <int> : <isa>

QUESTION FIVE: **[11 pts]**

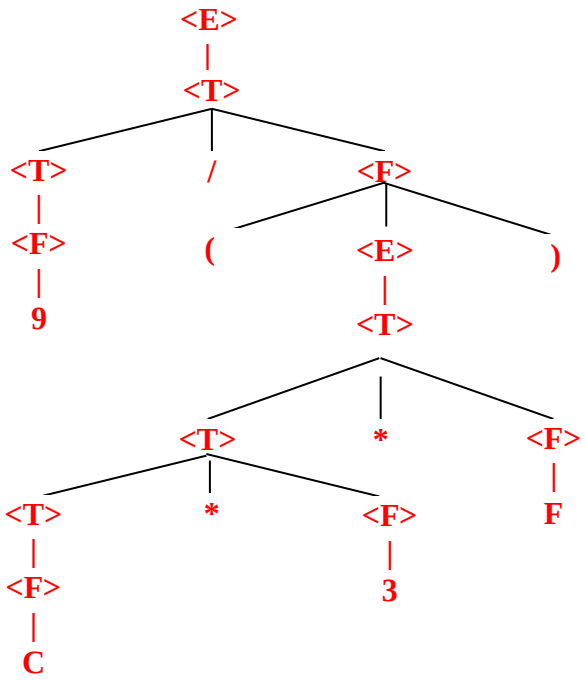
Carefully study the following grammar and answer the next two questions.

<E> → <E> + < T> | <E > – < T> | <T>
<T> → <T> * <F> | <T> / <F> | <F>
<F> → (<E>) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

- Construct the rightmost derivation of the sentence : 5 + (D * A + 9)

<E> => <E> + <T>
=> <E> + <F>
=> <E> + (<E>)
=> <E> + (<E> + <T>)
=> <E> + (<E> + <F>)
=> <E> + (<E> + 9)
=> <E> + (<T> + 9)
=> <E> + (<T> * <F> + 9)
=> <E> + (<T> * A + 9)
=> <E> + (<F> * A + 9)
=> <E> + (D * A + 9)
=> <T> + (D * A + 9)
=> <F> + (D * A + 9)
=> 5 + (D * A + 9)

- Construct the parse tree of the sentence : 9 / (C * 3 * F)



QUESTION SIX: **[12 pts]**

- Construct the BNF rules to define the **cout** statement defined as follows. A **cout** statement is a keyword **cout** or **print** followed by an insertion operator **<<** followed by one or more expressions separated by colons **:** and terminated by **$**. An expression may be one or more variables or constants. A constant is one or more digits. A digit is **<dig> →0|1|…|8|9**. Examples of accepted **cout** statements:

```
cout << <var>:345:<var> $
cout << 67 : <var> $
cout << <var> $
```

```
<cout>    → cout << <exprs> $ | print << <exprs> $
<exprs>   → <expr> | <expr> : <exprs>
<expr>    → <const> | <var>
<const>   → <dig> | <const> <dig>
```

- Construct the BNF rules to define the **cin** statement defined as follows. A **cin** statement is a keyword **cin** followed by the extraction operator **>>** followed by one or more identifiers separated by **colons :** and terminated by **$**. An identifier is defined as a letter followed by zero or more digits or letters. Assume letters and digits are already predefined. Examples of accepted **cin** statements:

```
cin >> quiz $
cin >> t : row4 $
cin >> d8 : h : b49d $
```

```
<cin>   → cin >> <ids> $
<ids>   → <id> | <id> : <ids>
<id>    → <let> | <id> <let>  | <id> <dig>
```